Distributed Database Access in the LHC Computing Grid with CORAL

Radovan Chytrcek, Dirk Düllmann, Giacomo Govi, Alexander Kalkhof, Zsolt Molnár, Andrea Valassi

Abstract—The CORAL package is the LCG Persistency Framework foundation for accessing relational databases. From the start CORAL has been designed to facilitate the deployment of the LHC experiment database applications in a distributed computing environment. In particular we cover - improvements to database service scalability by client connection management - platform-independent, multi-tier scalable database access by connection multiplexing, caching - a secure authentication and authorisation scheme integrated with existing grid services. We will summarize the deployment experience from several experiment productions using the distributed database infrastructure, which is now available in LCG. Finally, we present perspectives for future developments in this area.

I. INTRODUCTION

THE *Large Hadron Collider* (LHC) [1], launched officially on 11 September 2008 at CERN, Geneva represents a new era in the high energy physics. It is supposed to answer the most exciting, fundamental questions of the nature that would completely change how we see the world. The LHC itself is the most complicated scientific instrument that was ever constructed; during its construction we had to solve several technical, scientific and computing problems, such problems that we have never faced with earlier. Those solutions contribute to the global technology and industry as well, not only to the accelerator science and physics.

Among others, the LHC construction boosted the development of the grid technologies as well. It was an early observation that processing and handling the expected enormous amount of data (hundred petabytes over the lifetime of the experiments) that the different LHC experiments will produce would require a completely different approach. Not

only the amount of the produced and stored data was the challenge but the anticipated CPU requirements of the data analysis was so high, that it would have been impossible to construct a central, CERN based data analysis facility. It got clear then that the data processing had to be distributed between the contributing institutes, based on their own, existing infrastructure. Different institutes have different infrastructure, access policy, software solutions – and precisely this inherent inhomogeneity is the main issue that the grid technologies address. So, it was a natural choice that the LHC computing infrastructure should be grid based.

If the data analysis is distributed, the data itself must also be distributed. But there is inhomogeneity also in the data storing (database) infrastructure of the different institutes. Replicating, transferring and accessing the same data reliably between different database systems and across different security policies is one of the major issues that was solved by the special, customized implementation of the grid services, called the *LHC Computing Grid* (LCG) [2].

The inhomogeneity is not limited to space; it is in time as well. During the long lifetime of the LHC, the computing infrastructure, the computing models will most probably change. Therefore, LCG also focuses on the change management and maintainability for the overall experiment computing infrastructure, and it must be able to adapt quickly when such changes occur.

The LCG made the computing and database systems transparent, but there was another problem to be solved: the data analysis problem. The physical data is stored in tables of relational databases, accessing them requires special software components and queries formed in SQL. However, the existing analysis applications in the framework of LHC are written mostly in C++, and their implementation logic does not always follow the relational database access logic. In addition, the application developers and users normally do not want to use SQL, do not know the details of accessing a database server, and they want to integrate the data source seamlessly into the existing analysis application. Especially, they do not want to adapt their applications to the different SQL dialects, and to the different ways of database server access systems. Therefore, a programming language level *database transparency* became also necessary.

Manuscript received November 14, 2008.

Z. M. Author is with the Information Technology Department, European Organization for Nuclear Research (CERN), Geneva, CH-1211 Switzerland (Zsolt.Molnar@cern.ch). Member of IEEE, ACM.

R. C. Author was with the Information Technology Department, European Organization for Nuclear Research (CERN), Geneva, CH-1211 Switzerland (Radovan.Chytrcek@cern.ch).

D. D. Author is with the Information Technology Department, European Organization for Nuclear Research (CERN), Geneva, CH-1211 Switzerland (Dirk.Duellmann@cern.ch).

G. G. Author is with the Information Technology Department, European Organization for Nuclear Research (CERN), Geneva, CH-1211 Switzerland (Giacomo.Govi@cern.ch).

A. K. Author is with the Information Technology Department, European Organization for Nuclear Research (CERN), Geneva, CH-1211 Switzerland (Alexander.Kalkhof@cern.ch).

A. V. Author is with the Information Technology Department, European Organization for Nuclear Research (CERN), Geneva, CH-1211 Switzerland (Andrea.Valassi@cern.ch).

II. THE CORAL PROJECT

The *Common Relational Abstraction Layer* (CORAL) package [3] of the *LCG Persistency Framework* (LCGPF) [4] addresses the issues above: it implements a C++ framework for accessing data in relational databases, by providing an SQL-free API isolating the technology-specific database server accessing functionalities from the data analysis code.

The CORAL project started in 2004, as the *Relational Abstraction Layer* (RAL) of the Persistency Framework. Now, it provides relational database access for the *Pool of persistent Objects for LHC* (POOL) [5] and the *LCG Conditions Database Project* (COOL) [6] projects, as well as to some end-user applications directly. Those persistent objects are, for instance, the detector condition data: calibration, alignment and slow control data describing the state of a detector at the time of data taking.

POOL, COOL, CORAL became the common persistency framework of the LCG that can store multi-petabytes of experiment data and metadata in a distributed and grid-enabled way. It follows a component based architecture that is laid down in the LCG Blueprint RTAG report [7]. The architecture follows (as much as possible) a technology neutral approach: abstract component interfaces isolate the user code of the experiment framework from the particular implementation technologies. “Technology” means, for instance, database backend technology, security, authentication solutions, etc. Applications based on LCGPF do not depend on the implementation API, and do not directly depend on the implementation libraries (there is no link-time dependency). An implementation library is, for instance, the official *Oracle* or *MySQL* client library in case of database access systems. The component implementations are loaded at runtime instead, via a special plugin infrastructure. In that way, we achieved the quick adaptability requirement of the LCG: user code of a technology implementation does not depend on the technology, so technology change is seamless (requires no recompilation either). The approach enables self-consistent deployment of a subset of the full LCGPF, or subset of the CORAL library set as well.

CORAL, being part of LCGPF, also follows the above architecture. It concentrates on the relational database technology issues only: how to write/retrieve data, how to access/authenticate to the database servers, how to locate and access different database replicas. It is expected to be used by applications running on grid-enabled and distributed environment. In such environments, CORAL must cope with the following problems:

- physical database service indirection
- separating database connectivity from the authentication mechanisms
- enabling certificate-based authentication mechanisms
- enable client-side connection pooling and service failover
- enable client-side monitoring

But the following points are out of CORAL scope, so CORAL is not:

- a general purpose C++ connectivity library
- designed to accommodate all possible use cases for accessing relational data - CORAL is restricted only to the specific use cases of the LHC experiments
- system performing C++ object I/O with an RDBMS

The existing, general purpose C++ connectivity libraries on the market do not fulfill the requirements, that is why in LCGPF, the development of CORAL as a specific solution was necessary.

The C++ CORAL application developer does not need to know and issue SQL commands¹. The user does not need to be expert on all the possible database optimization techniques relevant to the particular database backend because they are implemented in the special database backend accessing plugins. But CORAL does not hide the fact that the data source is a relational database, so the user must be familiar with the basic concepts of the relational database systems such as *queries*, *sessions*, *transactions*, etc. On the other hand, the CORAL interfaces “guide” the user towards some standard “best practices” in RDBMS programming, such as *variable binding*, *result set pre-fetching*, *bulk operations*, etc.

The actual SQL commands are generated by the backend plugins. The generated commands are limited to a relatively small subset of the SQL command set that makes the client-side monitoring and tuning of the CORAL applications rather easy.

When this article was published, CORAL supported the following relational database technologies: *Oracle*, *MySQL*, *SQLite* *Frontier*.

CORAL has also plugins to support the different authentication and replica lookup services. For the details of those components, we refer to the online documentation of the CORAL package [8].

In the rest of the section, we demonstrate the above principles by a simple CORAL example. In Listing 1, we can see a typical CORAL code fragment that executes a `SELECT * FROM TABLE_NAME` query.

What happens in *line 3* demonstrates the main insulation principles of CORAL. Its input is an identification string that identifies a database schema. The string contains sufficient information to start a database session by using the *ConnectionService* module that is the entry point to the CORAL system. It executes the following main operations:

- The system loads a *database lookup plugin* that interprets the string. In the example above, by default, the system loads a plugin that reads XML files describing the database technology type, the

¹ With the exception of simple WHERE clauses

server location, the schema information, and the security/access schema details. According to the retrieved information, the component:

- Loads the corresponding *database access plugin*.
- Loads the plugin that handles the *security schema*.
- Establishes a connection to the database server, by calling the interface methods of the security schema handling plugin.
- Starts an updating database session, by using the access plugin interface.
- Returns a handle to the open database session. The handle completely hides the details above.

From *line 4*, the traditional database operating sequence starts. The code starts a database transaction, and defines a query, without injecting any SQL. The table is identified by its name in the database, and the user code accesses it as a table handle. As we do not specify any more details for the query, the executed SQL is the default `SELECT *`.

```
1. coral::ConnectionService cService;
2. string db = "/tests/db_ora/admin";

3. coral::ISessionProxy* session =
   connService.connect(db, coral::Update);

4. session->transaction().start();
5. coral::ISchema& schema = session.nominalSchema();

6. coral::IQuery* query =
   schema.tableHandle("TABLE_NAME").newQuery();

7. coral::ICursor& c = query->execute();

8. while (c.next())
9.     cout << c.currentRow["COLUMN"].data<float>();

10. session->transaction().commit();
```

Listing 1. Example CORAL code

As we can see then, CORAL is able to hide the technology details of a particular RDBMS, including the SQL dialect, the connectivity and the authentication, but it does not solve all the existing problems: we have to face with some deployment issues as well.

III. DEPLOYMENT SCENARIOS

In this section, we will investigate some deployment issues coming from the traditional, 2-tier client-server model of the CORAL application and database backend systems. Figure 1 demonstrates the basic deployment. The database servers are located behind the institute firewalls. CORAL applications may run outside of the firewall. A CORAL application includes the following main components:

- the data analysis logic
- the appropriate CORAL database backend access plugin, loaded in runtime
- the third-party database backend access client libraries linked to the plugin

Figure 1. 2-tier model of the CORAL application systems

The third-party libraries directly connect to the database servers, through the institute firewall. It means that the database servers must be visible to the Internet. This is normally a security issue, because malicious parties may exploit the database server vulnerabilities.

Direct database connection may conflict with the grid security policies as well. It is not ensured that the database servers support proxy certificates, the Grid Security Infrastructure (GSI), etc. Therefore, the system needs to use those basic authentication mechanisms that a given database server provides. In some setups, it may even involve transmitting plain text username-password pairs over the internet that is extremely dangerous.

Another deployment issue is the usage of the third-party libraries themselves. Those libraries must be installed and available on each client nodes. Sometimes, is not possible, due to the software licensing policies of a given institute: they do not want or cannot purchase or obtain the required software. It means that those institutes simply cannot use the LHC data.

The minimal deployment has some resource utilization and performance problems as well. Each client represents a physical connection to the database servers, and the core CORAL does not provide a mechanism for sharing several sessions from the same user application over a single physical connection. Sometimes, the clients reside on the nodes of a computing farm with ten thousands of nodes, and that is an enormous number of database connections. However, in the

LHC use cases, those physical connections are frequently idle, just holding precious computing resources on the server side.

Another resource sink comes from the fact that during some data analysis scenarios, many computing nodes require the same set of data. It means that they execute the very same database queries, and obtain the same result sets, but in several instances. It also takes expensive networking and database server resources.

The above problems have also been addressed by CORAL recently. The related developments started in the last quarter of 2007, as a joint project with an ATLAS [9] team at SLAC. In the followings, we overview the proposed solutions for the problems above.

The basis of the solution is that we introduce a multi-tier deployment model, see Figure 2. The figure demonstrates the deployment model containing all of the components - real deployment scenarios may be simpler.

Figure 2. Multi-tier model of the CORAL application systems

As we can see, clients may still connect directly to those database servers which access policies allow it, and the connectivity libraries may be installed properly. We can also see a new component, called *CORAL server*. The CORAL server resides behind the institute firewall, close to the database servers. The database servers are not visible to the internet, only the CORAL server is visible. The CORAL clients connect to the CORAL server only.

The CORAL server is a CORAL application that enables executing the CORAL operations remotely, on behalf of the clients. The CORAL operations refer to the database access plugin interface calls. The way of transferring the operation instructions from the client to the server and sending the results back is pure CORAL development.

Therefore, in the deployment model including the CORAL server, the client does not need to load the appropriate database access plugin, because the CORAL server does it, and executes the client operations against them. It solved the deployment problem: third-party database access libraries do not need to be present on the client side.

It also provides solution for the security problem. The CORAL server may utilize several plugins implementing security and access policies between the client and the servers, and it can translate them into database access policies. The only component that is visible toward the internet is the CORAL server: we have to cope only with the CORAL server vulnerabilities only, and the fixes are much quicker comparing to the response time of the maintainers of the database systems.

The solution involving the CORAL server has also some performance optimization, because CORAL itself implements a *connection pooling* feature: the established physical connections are not terminated immediately when the application logic requests it, so the already established connections may be reused. Idle connections may be reused by several client threads.

There is a connection pool for each CORAL client. The CORAL server is a CORAL application, behaves as one single CORAL client toward the database servers, so it also has its own connection pool. It means that it can multiplex several client connections into fewer physical connections toward the databases, using its connection pool.

We still need to face with the problem of executing the same database query several times. The *CORAL proxy* component is the proposed solution for the problem. The CORAL proxy is deployed close to the data analysis facilities (the users of the data). Clients connect directly to the proxy instead of the CORAL server. The proxy forwards the requests to the CORAL server, receives the reply – and *caches* them. If another client sends a request, the proxy may serve the reply from the cache. It means, for example, that in ideal case, the semantically different queries are executed only once on the database servers.

The proxy represents a single client to the CORAL server. The requests coming from the several clients are multiplexed into that single physical channel, so the CORAL server will open only one physical connection to the database servers.

From development point of view, there is another “database access plugin” that implements the connectivity and communication between the CORAL client and the CORAL server. In fact it is not a real database access plugin, because it connects to the CORAL server, but it implements the very same interfaces that those do. That particular plugin does not need to know whether the server it connects to be a CORAL server or a CORAL proxy. So for a CORAL application, accessing the CORAL server is transparent: it does not need to know if the code is executed against the database server, CORAL server or CORAL proxy.

IV. COMMUNICATION PROTOCOL

The communication between the CORAL servers and clients is based on a proprietary protocol, developed only for CORAL purposes. It is *message based*: a complete request is encapsulated into a message, the server receives and processes the message, then executes the appropriate CORAL operation against the database server. The results are sent back as response messages.

The protocol has two layers, a *transport* and a *relational* layer. The transport layer is implemented as message headers. The transport layer protocol is based on an earlier development called *DbProxy* [10], used by the ATLAS experiment group. The structure of the headers is illustrated by Figure 3.

Figure 3. CORAL transport protocol

The protocol is a binary protocol. It is able to handle the different computer architectures, from the endianness point of view. There is no fixed endianness for the messages: the systems construct the sent messages according to their natural endianness, and converts the received messages if they detect different endianness. The endianness detection is based on the very first (“magic”) word of the message headers; that word is also the message separator.

Each client is identified by a *client ID*. A client may be a CORAL application, a thread of a client application, etc. This is the base of the connection sharing and multiplexing: messages from different clients can be multiplexed into one physical channel. The individual request messages coming from a client are identified by the *request ID*.

The messages may even be segmented. A segment may be, for example, a given number of rows in a query result set. The segments have role in the proxy cache: segments of big queries may be cached individually. All the segments have transport layer headers, with segment numbers.

Not all the messages are cached. The error messages are examples of those messages that should not be cached. So, the message header contains a flag that controls if the message may be cached or not. The message *opcode* identifies the message type: the message type normally refers to a relational operation.

The header contains sufficient information for the proxy about where to forward a particular request, and fully implement the physical channel multiplexing, the client/request identification. The relational semantics of the payload is not analyzed by the proxy. In case of cacheable requests, the proxy takes the payload (or a fragment of the payload) as a binary cache key, to identify a reply in the cache. The payload of the requests is transferred to the CORAL client as is, from the cache. This enables extremely quick serving of the cached data. The relational data is fetched by the CORAL client, and this operation is faster than executing the corresponding query. So, not only we saved enormous computing resources, but the execution performance of the CORAL application is also better with the cache. According to the latest performance measurements, executing small, similar queries several times with the CORAL proxy provides significant performance gain comparing to the query execution directly on the database server.

The message payload represents the *relational layer* of the protocol. We have messages that represent *queries*, *schema metadata retrievals*, *connection*, *session*, *transaction operations*, etc. The messages rely on the semantics of the CORAL insulation interfaces, but they are not simple remote method calls: they “record” the interface calls, and message exchange occurs only when the interaction with the database servers cannot be delayed any more (lazy evaluation scheme).

Important aspect of the protocol is providing effective *caching*. We developed a method that is able to decide if two messages carry semantically the same relational information or not. In the equivalent cases, the generated cache keys are the same. For example, two messages carrying query information are semantically equivalent if they differ only in the order of the selected columns. The decision is made on the insulation interface method calls, we do not use some “proprietary” SQL or similar representation, because the set of interface method calls determine the relational operations unambiguously, on all the supported database backends. The details are described in the *CORAL Server Protocol Description* [11], available from the CORAL project web site.

The cache keys are generated on the client side, because the CORAL proxy does not analyze the message semantics. This approach has the benefit that the protocol changes are transparent to the proxy, in accord with the quick change management requirements of the LCG: the proxy does not need to be changed/re-deployed if CORAL technology changes. Another assurance of the quick adaptability is that the protocols are versioned at their lowest layer, so we took into account the backward compatibility issues as well.

V. SERVER-PROXY SOFTWARE COMPONENTS

Let us overview briefly the CORAL software components involved in the client-server-proxy scenario (Figure 4).

The *CoralProtocol* module implements the transport layer protocol. It is a thin software component, contains the header definitions and operations only. On top of this, *CoralMessaging* implements the full messaging system. It is responsible for *marshaling/un-marshaling* the messages, handles the physical network connection, provides a reliable messaging layer. *Reliable* means that the client code have to handle full, consistent, reconstructed messages only, but this module does not check the semantics.

All the messages have static structure, determined in compile time, only the data itself must be filled in runtime. By using extensive meta-programming techniques based on the *Boost Metaprogramming Library* [12], we are able to check the syntactical validity of all the message exchanges in compile time. Certainly, the messaging system still contains runtime syntactical checks, because syntactically incorrect messages may come up due to runtime software errors.

Figure 4. CORAL server and proxy software components

Due to this static nature, the same metaprogramming techniques can be used to generate the messaging code as well, that enables quick prototyping of new messages, and we can avoid a lot of human coding errors. We can define all the messages in a very simple, C macro-based embedded language, and the marshaling/un-marshaling code, the message classes, etc. are automatically generated. For example, in Listing 2, we can see the message responsible for queries for the table metadata.

The code in Listing 2 is sufficient to create a C++ class representing the message and the full class along with the integration to the system. The only code that must be explicitly written is the server-side message handler in the CORAL server module.

CoralMessaging is the module where multiplexing/de-multiplexing of the messages coming from several clients is

handled. The concept is similar to that of the networking sockets, where a TCP packet origin is identified by the IP address and the port number. The *CoralSocket* concept extends the TCP socket concept with the client and request ID-s, and the four pieces of data determine the CORAL message origin completely (that may be a thread of a CORAL application on a computing node, etc).

The *CoralAccess* module is an implementation of the CORAL plugin interfaces. It is the component that the client software loads in runtime, and behaves like an ordinary database backend plugin. At certain points of the software execution, it creates messages, exchanges it with the CORAL server (or proxy), by using the *CoralMessaging* module.

```

#define CAL_CLASSNAME TableSchemaRead
#define CAL_CLASSNAME_STR "CAL::TableSchemaRead"
#define CACHEABLE 1
#define CAL_PAR_NUM 2
#define CAL_PAR_NAME0 schemaName
#define CAL_PAR_NAME1 tableName
#include
"CoralMessaging/CAL/CALMessageGenerator.h"
CAL_MESSAGE_CLASSGENERATOR_2(TableSchemaRead,
std::string, std::string);
  
```

Listing 2. Defining messages

The *CoralServer* is a CORAL application receiving and handling the messages. It is a multi-threaded, thread pool utilizing network server, where each message represents an event. A connection *Dispatcher* listens to the network connection requests, and starts an *Event Dispatcher* in a separate thread. The Event Dispatcher receives the messages, and prepares a new thread of execution for each successfully received message. The thread then executes the relational operation encoded in the message, and sends back the result.

The messages arriving from a physical channel share the same database session/transaction context. This is the only state that the server maintains; the event handlers themselves are *stateless* and independent from each other. Therefore, the server is *well scalable*, in the sense of client connections. This paves the way for a network of servers with load balancing, if the demands grow high.

The server also handles the security data and the authentication schemes. The databases are not accessed until a client authenticates itself, according to the required policy. The authentication information is transmitted through a different (optionally secured) control channel. After successful authentication, the relational message exchange goes through the unsecured data channel. The data channel security is not a requirement: we want to protect our infrastructure; the data itself is not confidential.

The *CoralProxy* component implements the caching proxy server. This component is developed at SLAC, based on the principles discussed earlier.

VI. SUMMARY

The article presented CORAL, the relational database abstraction layer of the LCG Persistency Framework. CORAL concentrates on the software development and deployment aspects of accessing relational data in large scale, inhomogeneous, distributed computing environments, based on grid techniques. We saw, that in the LHC experiment scenarios, we are able to provide a collection of abstract interfaces that hides the database backend technology details from the application developers, doing it in an SQL-free way. CORAL also addresses several deployment and performance problems by introducing a multi-tier system of servers providing effective data caching, resource multiplexing and resource relocation. The tests and deployment of the CORAL based systems are ongoing; the first full setup will be available in the first half of 2009, in the framework of the ATLAS experiment.

ACKNOWLEDGEMENTS

We would like to thank *Rainer Bartoldus* (bartoldu@slac.stanford.edu) and *Andrei A. Salnikov* (salnikov@slac.stanford.edu) at SLAC for their contribution to the CORAL project including among others, the CORAL proxy server development, their contributions to the communication protocol and the testing.

To the Physics Database Service team at IT-DM at CERN and the SPI team the CERN LCG Application Area.

REFERENCES

- [1] LHC Web page. [Online] <http://public.web.cern.ch/public/en/LHC/LHC-en.html>.
- [2] LHC Computing Grid web page. [Online] <http://lcg.web.cern.ch/LCG>.
- [3] CORAL web site. [Online] <http://pool.cern.ch/coral>.
- [4] LCH Persistency Framework web page. [Online] <http://lcgapp.cern.ch/project/persist>.
- [5] POOL Web Site. [Online] <http://pool.cern.ch/>.
- [6] COOL Web Site. [Online] <http://cool.web.cern.ch/cool>.
- [7] LCG Bluepring RTAG Report. [Online] <http://lcg.web.cern.ch/LCG/sc2/RTAG1/>.
- [8] CORAL User Guide. [Online] <http://pool.cern.ch/coral/currentReleaseDoc/UserGuide/>.
- [9] ATLAS experiment web page. [Online] <http://atlas.web.cern.ch>.
- [10] ATLAS DbProxy Project Web Page. [Online] <https://twiki.cern.ch/twiki/bin/view/Atlas/DbProxy>.
- [11] Molnár, Zsolt. CORAL Server Protocol Description. Geneva : s.n., 2008.
- [12] Boost Metaprogramming Library. [Online] http://www.boost.org/doc/libs/1_37_0/libs/mpl/doc/index.html.